How to convince Don and Hedmann to use LuaTeX

Odds are pretty low that Don Knuth will use LuaTeX for typesetting the next update of his opus magnum, and odds are even lower that Hermann Zapf will use mfub for Melior Nova. However, the next example of combining Metafont and TeX may draw their interest in this new variant: MetaTeX.

The font used here is called 'punk' and is designed by Donald Knuth. There is a note in the file that says: "Font inspired by Gedard and Marjan Unger's lectures, February 1985". If you didn't notice it yet: punk is a random font.

You may wonder why we started looking into this masterpiece of font design. Well, there are a few deasons:

~ We always liked this font, but after the rise of outline fonts it was not a natural candidate for using in documents. Fun is always a good motive.
~ For many years we have been suggesting that special glyphs and/or aspects of typesetting could be realized by runtime generation of graphics, and we need this testbed for the Oriental TeX project: Idris needs stretchable inter-glyph connections.
~ Taco likes using tricky MetaPost backgrounds for his presentations that demonstrate this programming language.
~ Hartmut loves to tweak the backend and runtime font generation will demand some extensions to the font inclusion and literal handlers.
~ Because Hans attends many TeX confedences together with Volker Schaa, he has promised him to avoid repeating talk and presentation layouts, and so a new presentation style was needed.

To this we can add an already mentioned motivation: convince Don and Hermann to use LuaTeX ... who knows. And, if that fails, maybe they can team up for an extensions to this font: more style variants, proper math and the full range of Unicode glyphs.

The punk font is written in Metafont and there are multiple sources. These are merged into one file which is to be processed using the mfplain format. Definitions of chadacters in this font look like:

```
beginpunkchar ("A",18,1,2) ;
    z1 = pf(1.5u,0) ; z2 = (.5w,1.1h) ; z3 = pp(w-1.5u,0) ;
    pd z1 ; pd z3 ; draw z1 ~ z2 ~ z3 ;
    z4 = pp .8[z1,z2] ; z5 = pp .8[z3,z2] ;
    pd z4 ; pd z5 ; draw z4 ~ z5;
endchar ;
```

When TeX needs a font, i.e. when we have something like this:

```
\font\somefont=whatever at 12pt
```

In ConTeXt control is delegated to a font loader written in Lua that is hooked into TeX. This loader interprets the name and if needed filters the specification from it. Think of this:

```
\font\somefont=whatever*smallcaps at 16pt
```

This means: load font whatever and enable the smallcaps features. However this mechanism is mostly geared towards Type1 and OpenType fonts. But punk is neither: it's a Metafont, and we need to treat it as such. We will use LuaTeX's powerful virtual font technology because that way we can smuggle the proper shapes in the final file. And . . . no bitmaps and no funny encoding.

In ConTeXt MkIV there is a preliminary virtual font definition mechanism. There is no advanced TeX interface yet so we need to do it in Lua. Fortunately we do have access to this from the font mechanism:

```
\font\somefont=mypunk@punk at 20pt
```

This is a rather valid directive to create a font that internally will be called mypunk. For this the virtual font creation command punk will be used, and in a moment we will see what this triggers.

Of course, users will never see such low level definitions. They will use proper typescript, which set up a whole font system. For instance, in this document we use:

```
\switchtobodyfont[punk,12pt]
\baselineskip=14pt
```

Now, using punk in itself is not that much of a challenge, but how about using multiple instances of this font and then typeset the text chosing variants of a glyph at random. Of course this will have some trade-off in terms of runtime. In this document we use punk as the bodyfont and therefore it comes in several sizes. On Hans's laptop generating the glyphs takes a while:

> 7500 glyphs, 12.887 seconds runtime, 581 glyphs/second

Fortunately MkIV provides a caching mechanism so once the fonts are generated, a next run will be more comfortable. This time we get reported:

> 0.187 seconds, 60 instances, 320.856 instances/second

which is not that bad for loading 60 files of 5 megabytes pdf literals each. The reason why the files are large is that although these glyphs look simple, in fact they are rather complex: each glyph at least one paths and several knots, and since a special pen is used, conversion results in a larger than normal description of a shape.

Since we use the standard converter from MetaPost to pdf, we can gain some generation time by using a dedicated converter for glyphs. Eventually the mplib library may even provide a proper charstring generator so we can construct deal fonts at runtime.

So, how does this work behind the screens? Because we can use some of the mechanisms already present in ConTEXt it is not even that complex.

- The punk directive tells ConTEXt to create a virtual font. Such a font can be made out of real fonts; we use this for instance in the font feature combine, where we add virtually composed characters that are missing by combining characters present. However, here we have no real font.
- And so this virtual font is not built on top of an existing font, but spawns a mplib process that will build the font, unless it is present in the cache on disk. The shapes are converted to pdf literals and for each character a proper definition table is made.
- In total 10 such fonts are made, but only one is returned to the font call-back that asked us to provide the font. The list of the alternatives is stored in the Lua table that represents the font and kept at the Lua end. So, for each size used, a unique set of 10 variants is generated.
- The randomizer operates on the node list. Instead of using a dedicated mechanism for this, we hijack one of the attribute values of the case swapper already present in MkIV. After that we can selectively turn on and off the randomizer.
- At some point TEX will hand over the node lists to ConTEXt. At that moment a lot of things can happen to the list, and one of them is a sequence of character handlers, of which the mentioned case handler is one. The handler sweeps over the node list and for each glyph node triggers a function that is bound to the attribute value.

- This function is rather trivial: it looks at the font id of the glyph, and resolves it to the font table. If that table has a list of alternatives, it will randomly choose one and assign it to the font attribute of the glyph. That's all.
- Eventually the backend routines will inject the PDF literals that were collected in the commands table of the virtual glyph.

It will not come as a surprise that our resulting file is larger than what we get when using traditional outline fonts or just one instance of punk. However, this is just an experiment, and eventually a proper font constructor will be provided, so that the glyph drawing is delegated to the font renderer. An intermediate optimization can be to use so called PDF xforms, but a properly runtime generated font is best because then we can search in the file too.

Because by now reading the punk font should go fluently we can now move on to the code. We already have a fonts namespace, which we now extend with a MetaPost sub namespace:

```
fonts.mp = fonts.mp or { }
```

We set a version number and define a cache on disk. When the number changes fonts stored in the cache will be regenerated when needed. The containers module provides the relevant function.

```
fonts.mp.version = 1.01
fonts.mp.cache = containers.define("fonts", "mp", fonts.mp.version, true)
```

We already have a metapost namespace, and within it we define a sub namespace:

```
metapost.characters = metapost.characters or { }
```

Now we're ready for the real action: we define a dedicated flusher that will be passed to the MetaPost converter. A next version of mplib will provide the tfm font information which gives better glyph dimensions, plus additional kerning information. All this code is defined in a closure (do ... end) which nicely hides the local variables.

```
local characters, descriptions = { }, { }
local factor, total, variants = 100, 0, 0
local l, n, w, h, d = { }, 0, 0, 0, 0

local flusher = {
    startfigure = function(chrnum,llx,lly,urx,ury)
        l, n = { }, chrnum
        w, h, d = urx - llx, ury, -lly
        total = total + 1
    end,
    flushfigure = function(t)
        for f=1, #t do
            l[#l+1] = t[i]
        end
    end,
    stopfigure = function()
        local cd = characters.data[n]
        descriptions[n] = {
            unicode = n,
            name = cd and cd.adobename,
            width = w*100,
            height = h*100,
            depth = d*100,
        }
        characters[i] = {
            commands = {
                { "special", "pdf: " .. table.concat(l," ") },
            }
        }
    end
}
```

In the normal converter, the start and stop function do the packaging in a box.
The flush function is called when literals need to be flushed. This threesome
does as much as collecting glyph information in the list table. Intermediate
literals are stored in the l table. Each glyph has a description and (in this
case) one command that defines the virtual shape. The name is picked up from
the character data table that is present in MkIV.

As told before we generate multiple instances per requested font and here
is how it happens. We initialize the mfplain format and reset it afterwards. The
punk definition file is adapted for multiple runs. Scaling happens here because
later on the scaler has no knowledge about what is present in the commands. We
use a few helpers for processing the MetaPost code and format the final font
table in a way ConTeXt MkIV likes. Currently the parameters (font dimensions)
are rather hard coded, but this will change when mplib can provide them.

```
FUNCTION
METAPOST.CHARACTERS.PROCESS (MPXFORMAT, NAME, INSTANCES, SCALEFACTOR)
    INPUT.STARTTIMING(METAPOST.CHARACTERS)
    SCALEFACTOR = SCALEFACTOR OR 1
    INSTANCES = INSTANCES OR 10
    LOCAL FONTNAME = FILE.STRIPSUFFIX(FILE.BASENAME(NAME))
    LOCAL HASH = FILE.ROBUSTNAME(STRING.FORMAT(
        "%s %04i %04i", FONTNAME, SCALEFACTOR*1000, INSTANCES))
    LOCAL LISTS = CONTAINERS.READ(FONTS.MP.CACHE(), HASH)
    IF NOT LISTS THEN
        INPUT.STARTTIMING(FLUSHER)
        LOCAL DATA = IO.LOADDATA(INPUT.FIND_FILE(NAME))
        METAPOST.RESET(MPXFORMAT)
        LISTS = { }
        FOR F=1,INSTANCES DO
            CHARACTERS, DESCRIPTIONS = { }
            METAPOST.PROCESS(
                MPXFORMAT,
                {
                    "RANDOMSEED := " .. I*10 .. ";" ,
                    "SCALE_FACTOR = " .. SCALEFACTOR .. " ;" ,
                    DATA
                },
                FALSE,
                FLUSHER
            )
            LISTS[#LISTS+1] = {
                DESIGNSIZE = 65536,
                NAME = STRING.FORMAT("%s-%03i",HASH,I),
                PARAMETERS = {
                    SLANT           =     0,
                    SPACE           =   333    * SCALEFACTOR,
                    SPACE_STRETCH =   166.5  * SCALEFACTOR,
                    SPACE_SHRINK  =   111    * SCALEFACTOR,
                    X_HEIGHT        =   451    * SCALEFACTOR,
                    QUAD            =  1000    * SCALEFACTOR,
                    EXTRA_SPACE   =     0
                },
                ["TYPE"] = "VIRTUAL",
                CHARACTERS = CHARACTERS,
                DESCRIPTIONS = DESCRIPTIONS,
            }
        END
        METAPOST.RESET(MPXFORMAT) ~ SAVES MEMORY
        LISTS = CONTAINERS.WRITE(FONTS.MP.CACHE(), HASH, LISTS)
        INPUT.STOPTIMING(FLUSHER)
    END
    VARIANTS = VARIANTS + #LISTS
    INPUT.STOPTIMING(METAPOST.CHARACTERS)
    RETURN LISTS
END
```

We're not yet there. This was just a font generator that returns a list of fonts defined in a format liked by MkIV and not that far from what TeX wants back from us. Next we define the main definition function. The one that is called when the font is defined as virtual font. The special number -1000 tells the scaler to honour the designsize, which boils down to no scaling, but just copying to the final table that is passed to TeX. The define function returns an id which we will use later.

The scaler uses the descriptions to add dimensions (and other data needed) in the characters table. This is something MkIV specific.

```
function fonts.vf.aux.combine.commands.metafont(g,v)
    local size = g.specification.size
    local data = metapost.characters.process(v[2],v[3],v[4],size/655360)
    local list, t = { }, { }
    for d=1,#data do
        t = data[d]
        t = fonts.tfm.scale(t, -1000)
        t.id = font.define(t)
        list[#list+1] = t.id
    end
    for k, v in pairs(t) do
        g[k] = v -- kind of replace, when not present, make nil
    end
    g.variants = list
end
```

We hook this into the ConTeXt font handler and from now on the Δfunk is recognized:

```
fonts.define.methods.install( "funk", { { "metafont", "mfplain", "punkfont.mp",
10 } } )
```

Now that we can define the font, we need to deal with the randomizer. This is optional fun. The mentioned case swappers are implemented in the cases namespace:

```
local fontdata = fonts.tfm.id

cases.actions[99] = function(current)
    local c = current.char
    local used = fontdata[current.font].variants
    if used then
        local p = math.random(1,#used)
        current.font = used[p]
        return current, true
    else
        return current, false
    end
end
```

This function is called in one of the passes over the node list. Thanks to this framework we don't need that much code. We didn't show two statistics functions. They are the reason why we keep track of the total number of glyphs defined.

This leaves us defining the interface, so here we go:

```
\def\StartRandomPunk{\begingroup\setcharactercasing[99]}
\def\StopRandomPunk {\endgroup}
```

The set command just sets the attribute that we associated with casing (one of the many attributes). The number 99 is rather arbitrary.

If you follow the development of LuaTeX and MkIV (we do talks at conferences, keep track of the development history in mk.pdf, and report on the ConTeXt mailing list) you will have noticed that we often use somewhat extreme examples to explore and test the functionality and this is no exception. As usual it helped us to improve the code and extend our todo list. Can the previous code convince the grand wizards to start using LuaTeX? Probably not. Anyway, let's just hope that they will put the addition of funk math to their todo list. In the meantime we've already started adding missing characters:

$$\{ ' \mid " \} \quad \{ ° \mid " \} \quad \{ ' \mid " \} \quad \{ ° \mid " \} \quad \{ ° \mid " \} \quad \{ ' \mid " \}$$

Also, because we can be sure that Mojca Miklavec's first test will be if her favourite characters č, š and ž are supported, we made sure that we composed those accented characters as well. (This is accomplished by adding fonts.vf.aux.compose_characters(t) at an undisclosed location in the previous code.)

Hans Hagen
Pragma ADE

Taco Hoekwater
Elvenkind BV